A Simple and Correct Even-Odd Algorithm for the Point-in-Polygon Problem for Complex Polygons

Michael Galetzka[1] and Patrick Glauner[2]
[1]*Disy Informationssysteme GmbH, Ludwig-Erhard-Allee 6, 76131 Karlsruhe, Germany*
[2]*Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, 4 rue Alphonse Weicker, 2721 Luxembourg, Luxembourg*
galetzka.michael@gmail.com, patrick.glauner@uni.lu

Keywords: Complex Polygon, Even-Odd Algorithm, Point-in-Polygon.

Abstract: Determining if a point is in a polygon or not is used by a lot of applications in computer graphics, computer games and geoinformatics. Implementing this check is error-prone since there are many special cases to be considered. This holds true in particular for complex polygons whose edges intersect each other creating holes. In this paper we present a simple even-odd algorithm to solve this problem for complex polygons in linear time and prove its correctness for all possible points and polygons. We furthermore provide examples and implementation notes for this algorithm.

1 INTRODUCTION

At a first glance, the point-in-polygon problem seems to be a rather simple problem of geometry: given an arbitrary point Q and a closed polygon P, the question is whether the point lies inside or outside the polygon. There exist different algorithms to solve this problem, such as a cell-based algorithm (Zalik and Kolingerova), the winding number algorithm (Hormann and Agathos, 2001) or the even-odd algorithm (Foley *et al.*, 1990) that is used in this paper. The problem is not as trivial as it seems to be if the edges of the polygon can intersect other edges as seen in Figure 1. This kind of polygon is also often called a complex polygon since it can contain "holes".

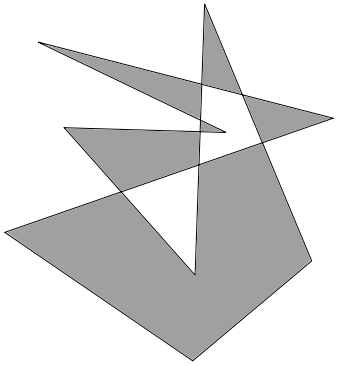


Figure 1: A self-intersecting polygon.

A lot of special cases have to be considered (e.g. if the point lies on an edge) and most of the existing even-odd algorithms either fail one or more of these special cases or have to implement some kind of workaround (Schirra, 2008). The rest of this paper is organized as follows. In Section 2, we present an even-odd algorithm and prove its correctness for all possible points and polygons with no special cases to be considered. In Section 3, we apply this algorithm to an example complex polygon. We then provide implementation notes in Section 4. Section 5 summarizes this work.

2 THE EVEN-ODD ALGORITHM

The basic even-odd algorithm itself is fairly simple: cast an infinite ray from the point in question and count how many edges of the polygon the ray intersects (Foley *et al.*, 1990).

Definition Let P be a polygon with n vertices $P_1, ..., P_{n-1}, P_n$ where the vertices are sorted in such a way that there exists an edge between P_i and P_{i+1} for every index $1 \leq i < n$ and between P_1 and P_n. The algorithm computes if the arbitrary point Q lies either *inside* or *outside* of P.

Galetzka, M. and Glauner, P.
A Simple and Correct Even-Odd Algorithm for the Point-in-Polygon Problem for Complex Polygons.
In *Proceedings of the 12th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2017) - Volume 1: GRAPP*, pages 175-178
ISBN: 978-989-758-224-0

The Steps of the Algorithm. To ease the presentation of the steps of the algorithm, it defines its own coordinate system with $(0|0) = Q$ by moving the polygon. This translation is only for illustration purposes. The algorithm then uses the positive x-axis as ray to calculate intersections with the edges of P. So the start vertex of the ray is $(0|0)$ and the end vertex is $(x_{max}|0)$ where x_{max} is an x value greater than that of any of the vertices of P.

1. First it is determined if Q is equal to any of the vertices of P or lies on any of the edges connecting the vertices. If so the result is *inside*.
2. A vertex P_s that does not lie on the x-axis is searched in the set of vertices of P. If no such vertex can be found the result is *outside*.
3. Set i to 1. Beginning from that vertex P_s the following steps are repeated until all vertices of P have been visited:
   (a) The index s is increased to $s+i$ until the next vertex P_{s+i} not lying on the x-axis is found. If the index $s+i > n$ then i is set to $-s$ and the search is continued.
   (b) Depending on the course of step (a) one of the following steps is taken:
      i. *No vertex has been skipped*: the line segment from P_s to P_{s+i} is intersected with the positive x-axis.
      ii. *At least one vertex with a positive x-value has been skipped*: the line segment from P_s to P_{s+i} is intersected with the complete x-axis.
      iii. *At least one vertex with a negative x-value has been skipped*: nothing is done.
   (c) P_{s+i} is the starting vertex for the next iteration.
4. If the count of intersections with the x-axis is even then the result is *outside*, if it is odd the result is *inside*.

Proof of Correctness. To prove that the oven-odd algorithm in general is correct one can generalize it to the Jordan Curve Theorem and prove its correctness (Tverberg, 1980). What has to be proven is that the given algorithm is in fact a correct even-odd algorithm, meaning that the count of edges is correct under all circumstances.

A lot of challenges emerge when trying to intersect two line segments and one of the segments has one or two of its vertices lying on the other segment. If the count of intersections with the x-axis are to be counted correctly, then each edge of P must either clearly intersect the x-axis or not intersect it at all. There must be no case where the starting or end vertex of a line segment lies on the line segment it should be intersected with.

The x-axis is the first line segment to look at, since it is part of all the intersections. The start vertex of the x-axis, namely Q, is guaranteed not to lie on any edge or to be equal to any vertex of P. If this was the case, then the first step of the algorithm would have already returned the correct result. The end vertex is guaranteed to have an x value greater than any of the vertices of P, so no vertex of P can be equal to it and no edge of P can contain it.

Of course there still exists the challenge that one or more vertices of P lie on the x-axis as seen in Figure 2.

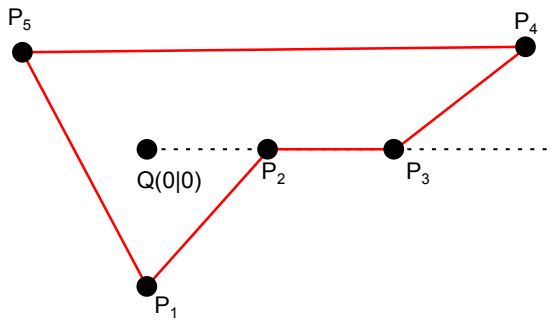


Figure 2: One of the edges of P lies on the x-axis.

The algorithm deals with this kind of challenge by ignoring the vertices lying on the x-axis and stepping over them when trying to find an edge to intersect. It then creates a new auxiliary edge that it can intersect safely. So starting for example at vertex P_1 it would ignore P_2 and P_3 and then create a new edge between P_1 and P_4 as shown in Figure 3.

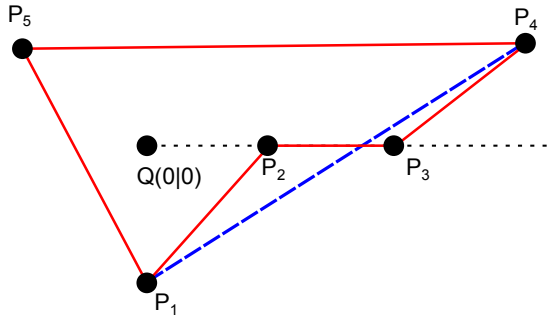


Figure 3: A new auxiliary edge has been created.

At this point it is clear that none of the edges used to calculate the count of intersections with the x-axis has a vertex on the x-axis and is either clearly intersecting it or not. What has to be shown is that all created auxiliary edges are correct substitutes to calculate the intersections with the x-axis.

Indeed they are not a correct substitute to intersect the positive x-axis, as can be seen in Figure 4. Here

the auxiliary edge would be the same as the edge between P_1 and P_3 and this edge does clearly not intersect the positive x-axis. So the total count of intersected edges of the polygon shown in Figure 4 would be zero - which is obviously wrong.

The algorithm actually deals with this challenge in step 3.*b*, by extending the ray in that special case where a vertex lying on the positive x-axis has been skipped. The new ray is then the complete x-axis and not only the positive part of it. It can be seen that this would create the desired result in the example of Figure 4, because the total count of intersected edges would then be one.

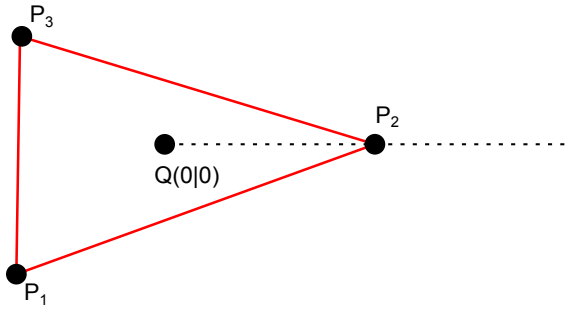


Figure 4: The auxiliary edge between P_1 and P_3 does not intersect the positive x-axis.

The question remains if this method always yields the correct result under all possible circumstances. If the skipped vertex lies on the negative x-axis then the auxiliary line will not be considered for intersection, since none of the original edges could have intersected the positive x-axis. So all cases that have to be looked at involve one or more vertices on the positive x-axis. The skipped vertices can never change from the positive to the negative x-axis since this would have been caught in the first step of the algorithm.

So, all auxiliary edges that intersect the x-axis are created from the following order of vertices: P_u which does not lie on the x-axis, P_v[1] which does lie on the positive x-axis and P_w which does not lie on the x-axis. Each of the vertices P_u and P_w can lie in one of the four quadrants around the point Q and therefore there exist $4 \times 4 = 16$ different versions of the auxiliary edge that have to be considered. This number can be further reduced to 10 because six of these versions are created by switching start and end vertex and do not have to be considered as a separate case.

All possibilities of the locations of P_u and P_w are shown in Table 2 as well as the desired intersection count and the actual intersection count of the algorithm. It is clear by looking at the table that the algorithm satisfies the desired result for each possible scenario. Therefore the auxiliary line is indeed a correct substitute for the original edges.

This leads to the conclusion that the algorithm provided can correctly determine if there is an even or an odd number of intersections with the polygon. □

Time Complexity. All n vertices of the polygon are visited once during the translation and the first three steps of the algorithm. All other computations like step four can be completed within constant time. Therefore the time complexity for this algorithm is $O(n)$. □

3 EXAMPLE

The algorithm is applied to the sample complex polygon P in Figure 5.

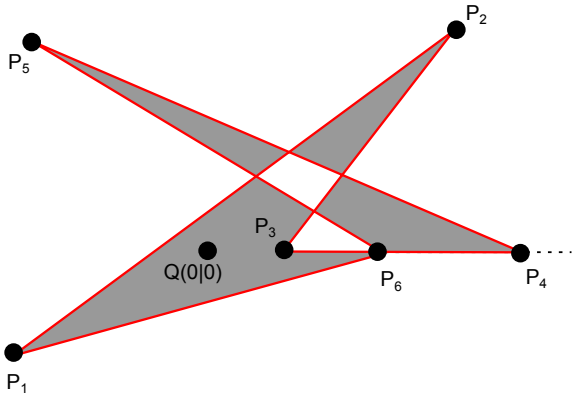


Figure 5: Sample complex polygon P.

1. Q lies neither on any vertex nor edge.
2. The start vertex P_s is P_1 in this example.
3. All intersection and substitution steps can be found in Table 1.
4. Since the count of intersections with the x-axis is odd the result is *inside*.

Table 1: Intersection and substitution steps.

P_u	P_v	P_w	x-axis for intersection operation	Inters.
P_1	–	P_2	positive	no
P_2	(P_3, P_4)	P_5	complete	no
P_5	P_6	P_1	complete	yes

[1]Of course there could be more than just one vertex on the x-axis but they are all skipped and the same auxiliary edge is created no matter how many additional vertices are on the x-axis.

4 IMPLEMENTATION

The translation of P can be done together with the first step of the algorithm by moving the vertices by the negative x- and y-values of Q. The geometric helper functions to intersect line segments can be found in (Stueker, 1999) which are an improved version of (Sedgewick, 1992). Our reference implementation of the even-odd algorithm for complex polygons presented in this paper is available as open source: http://github.com/pglauner/point_in_polygon.

5 CONCLUSIONS

The implementation of point-in-polygon algorithms for complex polygons is error-prone since there are many special cases to be considered. We presented a correct even-odd algorithm that prevents special cases by substituting a sequence of edges with an auxiliary edge. The correctness of the algorithm has been proven for all possible polygons, including complex polygons. The algorithm is also time-efficient since it is $O(n)$ for polygons with n vertices.

REFERENCES

Foley, J. D., van Dam, A., Feiner, S. K. and Hughes, J. F. (1990). Computer Graphics: Principles and Practice. The Systems Programming Series. 2nd Edition. Addison-Wesley, Reading.

Hormann, K. and Agathos, A. (2001). The point in polygon problem for arbitrary polygons. *Computational Geometry*, 20(3):131–144.

Schirra, S. (2008). How reliable are practical point-in-polygon strategies? *Algorithms - ESA 2008*, 5193:744–755.

Sedgewick, R. (1992). Algorithms in C++. 1st Edition. Addison-Wesley Professional.

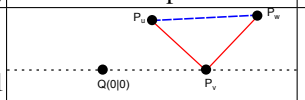
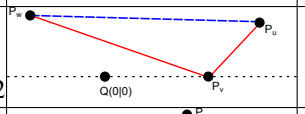
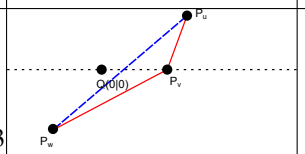
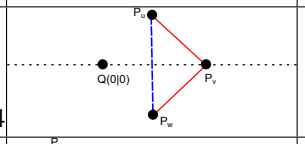
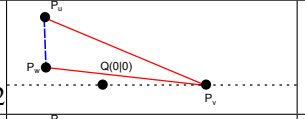
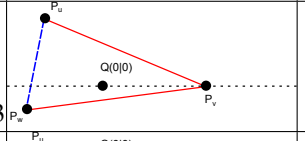
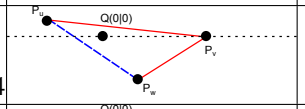
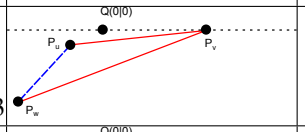
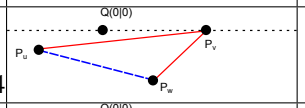
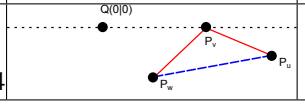
Stueker, D. (1999). Line segment intersection and inclusion in a polygon. *Elementary geometric methods*.

Tverberg, H. (1980). A proof of the jordan curve theorem. *Bulletin of the London Mathematical Society*, 12(1):34–38.

Zalik, B. and Kolingerova, I. (2001). A cell-based point-in-polygon algorithm suitable for large sets of points. *Computers and Geosciences*, 27(10):1135 – 1145.

APPENDIX

Table 2: Possible cases when creating an auxiliary line (blue). The values $q1$ to $q4$ correspond to the quadrants of a cartesian coordinate system ($q1 : x > 0 \wedge y > 0, q2 : x < 0 \wedge y > 0, ...$). Rows that are just gained by switching the vertices P_u and P_w are omitted due to symmetry.

P_u	P_w	Example	Desired count	Inters.
$q1$	$q1$		even	no
$q1$	$q2$		even	no
$q1$	$q3$		odd	yes
$q1$	$q4$		odd	yes
$q2$	$q2$		even	no
$q2$	$q3$		odd	yes
$q2$	$q4$		odd	yes
$q3$	$q3$		even	no
$q3$	$q4$		even	no
$q4$	$q4$		even	no